The evolution of CACSD tools-a software engineering perspective

Ravn, Ole; Szymkat, Maciej

Published in:
IEEE Symposium on Computer-Aided Control System Design

Link to article, DOI:
10.1109/CACSD.1992.274428

Publication date:
1992

Document Version
Publisher's PDF, also known as Version of record

The Evolution of CACSD Tools - A Software Engineering Perspective

Ole Ravn,
Institute of Automatic Control Systems,
Technical University of Denmark, Building 326,
DK-2800 Lyngby, Denmark

Maciej Szymkat,
Institute of Automatics,
Academy of Mining and Metallurgy, al. Mickiewicza 30,
30-059 Cracow, Poland.

Abstract

The paper presents the earlier evolution of CACSD tools in an software engineering perspective. A model of the design process is presented as the basis for principles and requirements of future CACSD tools. Combinability, interfacing in memory and an open workspace is seen as important new concepts in CACSD. Some points are made about the problem of "buy or make" when new software is required. The idea of "buy and make" is put forward. Emphasis is put on the time perspective and the life cycle of the software.

1 Introduction

Starting in the late fifties and early sixties computers available to control system designers were not very powerful, nor very easy to program or interact with. Features of software packages for Computer Aided Control Engineering have been determined by the commonly available computer technology at that time.They were used to automate difficult numerical calculations such as FFT. Most of the work done to design a controller was still done manually. During the sixties batch-operation of large computers became commonly available and made it feasible to use computers for tasks such as calculation of frequency responses, time responses, root loci and simulation. These programs were made at departments and universities all over the world as there were no commercial programs available. The resulting programs and libraries were in many cases incompatible what made it difficult to reuse data in other programs.

LINPACK and EISPACK were introduced in the mid-seventies providing well-tested solutions to various numerical problems. They became the basis for more specialized libraries for solving control problems. The designer still had to write the main program and this was often a difficult task as data structures had to be adjusted. Debugging was also difficult.

The introduction of terminal-access to mainframe computers made it possible to make interactive programs which could perform many different tasks, analysis, design and simulation on data from the same program. A number of commercial programs of this kind emerged in the seventies. These programs were often very large and at the same time difficult to extend. Many designers still found it easier write their own software.

In 1981 MATLAB was introduced. The formula of MATLAB is relatively simple:

MATLAB:=

LINPACK/EISPACK + Command interpreter + Graphics

MATLAB solved many of the compatibility and software development problems. The control system designer may access through MATLAB many tools that can be used in the design process. It is normally not as time consuming to make the necessary specific software for solving a problem in MATLAB as before. However MATLAB has certain limitations, too. One being the simple data structure (i.e. a complex matrix), another being the large number of m-files. A number of 300-400 m-files is easily reached when using some of the extra tool-boxes in MATLAB. It can be more difficult to find the m-file that solves the problem than writing a new one. This implies that some data-management system with more advanced typing and a tool-management system is needed. A number of packages with features similar to MATLAB and better interface has emerged, but none has gained the same wide use as MATLAB.

The introduction of workstation and the extended availability of powerful graphics computers has made it possible to meet new objectives. A number of graphical interfaces embedding MATLAB and some simulation package has emerged. These packages hide the MATLAB interface and provide some sort of graphical front-end as well as some data management. However this is often accomplished at the expense of the easy extendability that characterizes MATLAB.

Mathworks has introduced a package called SIMULAB which extends MATLAB with a non linear simulation tool while preserving the design and analysis capabilities of MATLAB. This is a step in the direction of more integrated environments.

Recently Integrated Systems Inc has introduced a new environment called Xmath featuring an object oriented approach to CACSD tools.

We would like to drive special attention to those aspects of CACSD tools development which can be addressed from the point of view of software engineering. The computer science perspective is certainly needed to understand the role of such factors as program portability, reusability or extendability.

It should be emphasized that the implementational issues do not cover the whole software development cycle. In what follows it will be assumed that the functional specification of the software requirements is already given, what means in the context of CACSD, that the design strategy, algorithms to be used, user interaction requirements are predefined in the conceptual phase of the design process.

The typical practical problems arising at the implementation stage are following :

- should the commercially available software be acquired ?
- should some algorithms be coded, who should do it and how ?
- should the available tools be used in a wider framework and what platform should used for the integration ?

It is important to say here that it would be naive to expect that some general answers for these questions could be given. Each particular situation has its own context and the feasible solutions have to take it into account. The aim of this contribution is to identify factors which constitute this context and essentially influence the software project development.

There are not many publications taking software engineering point of view on CACSD, just to mention here [2], [21] only. Certainly it is much more attractive to write how the "ideal tool" should look like, than to report why this or that project turned out to be unsuccessful. On the other hand in the field of the CACSD software we meet a very rapid "depreciation" process, the programs and packages written five years ago, may look today quite obsolete just because of the fast changes in user-interfaces and software development tools available at the moment, and it may seem that it is not very much worthy to deal with the "old program" any more. What we found very important is that some time perspective is necessary to assess to whole software development project.

The preferred solution for the implementation of the CACSD software is acquiring the existing commercially available products. If it is possible to find the product which is compatible with the available hardware and software resources, which fits well to the design requirements, which is reliable and well supported by the vendor, assures good portability for expected future applications, there is no doubt that the solution of this kind will probably be the most cost-efficient. Unfortunately, almost never all these conditions are fully satisfied and it is quite acceptable if only some adoption or integration is needed.

Two remarks are here in turn. Although, it is generally not advisable to write anew the "original" code for matrix manipulation or nonlinear simulation, there may exist situations when some reasonably small improvements in the already available software may be attractive, because the user does not have to change his habits and waste time for training to adopt the new tool.

The second remark concerns some potential drawbacks of acquiring the ready-made software which may include :

- *Lack of reliable software quality measurment tools* — the use of existing benchmarks, c.f. [7], [17], is hardly sufficient for comprehensive assessment,
- *hidden limitations* — meaning a kind of the "closedness" of the tool concept that may make it useless for future applications,
- *dependence on the vendors development policy* — e.g. the lack of future support for new the hardware and system software platforms.

It may be proved on many examples that vendors of successful software products in the product-oriented (not user-oriented) software market try to use their monopolist-like position, simply because they do not have enough incentives for the substantial improvement of their products, they introduce not much different new versions just to keep the user in a kind of trap. The distributed users (user groups) cannot force the real changes. The only positive thing here is the competition among the leading vendors.

2 CACSD and scientific and engineering computing

Let us invoke the early definition of CACSD cited in c.f. [20]

> *the use of digital computers as primary tool during the modelling, identification, analysis and design phase of control engineering*

and think if it should be today updated. How many other tools of control system design are used — e.g. paper and pencil, pocket calculator ? If we separate the conceptual phase belonging rather to mathematical control theory domain, there is almost no other tools except for computers, unless we do not exclude the use of data acquisition cards with AC/DC converters, analog simulators and maybe some more electronic equipment used for prototype installations. This is why there is a strong feeling that the computers are now a standard tools for control design and that there is nothing special about it. Even more, computers have overwhelmed the whole control engineering and that is why it seems to be more appropriate now to talk about the *Computer Aided Control Engineering.*

Certainly, there are some specific problems that solely concern the man-machine interaction, the use of human and "artificial" intelligence in the control design process, which are very much specific to CACSD, see the inspiring paper [13]. In some way, they also relate to the software development process. It is important to note here that these problems are in some sense common to other engineering disciplines. The interesting view from that perspective is presented in [10].

It is well known that CACSD software is only the part of wider class of computer software supporting the so called *science and engineering computing.* This class is the second to data processing computing in terms of general use of computers, where the software engineering methodology is already well established. What we are witnessing in recent years is that also in the scientific/engineering software there is a product-oriented software technology with a typical market containing specialized companies (the software vendors) and clients (the software users). However, so far there was little said on specific scientific/engineering software development paradigm. May be the reason is the extremely fast "software depreciation" in that area.

This may be explained also by the fact that the driving force of the technological progress in the computer domain is surely the *hardware* which doubles its (installed) capacities more or less every 2–3 years, the *software* can hardly keep this pace, with doubling time of 6–12 years, the whole *computing paradigm* concerning the way of the use of computer technology in other fields is much slower with very rough estimate of 15–25 years doubling time. The author of [4], see the technology comparison table on p. 94, claims that this process is caused by the modularity in the design and reusability of independent components used in the hardware development, but not in the software development where the new tools were introduced without increasing the units

of modularity, what had to result in exponential growth in hardware technology, but only arithmetic growth in software technology. It may be a good explanation but one has to remember also that software is a secondary factor in the computer technology and it is very hard if not impossible to predict how the future developments in hardware could affect the software. Anyway it is good to keep in mind this general time perspective.

The whole thing is here surely oversimplified because it is of course unrealistic to treat the software as the one group, neglecting its diversification into system level software, general purpose programming languages, specialized packages and environments, and application programs, similarly the hardware is not a homogeneous object.

3 The Structure of the Design Process

It is not feasible just to automate analysis methods any more. A look at the structure of the design process for controller design gives insight to objectives that could be pursued in future CACSD packages.

The design process for a control system is very complex. Some work has been done to model this process in a CACSD framework, c.f. [13]. The model presented below is a simplified model which only has a fraction of the possible interactions. Many more cross-interactions are present but for the purpose of pointing out possible objectives for CACE systems this model is sufficient. The implementation aspect of controller design is not addressed with this model. The model is shown in figure 1.

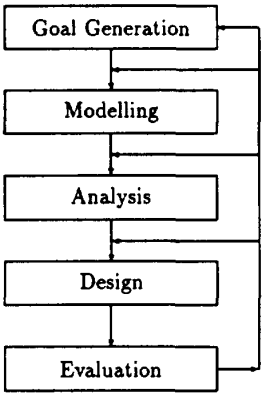

Figure 1: Structure of the Design Process.

The model contains 5 phases. The first phase to be entered when the design process starts is the problem description and goal generation phase. In this phase the definition of the design goal is generated and the control problem is specified. The second state is the modelling phase. Often a linear model in an appropriate domain will be generated during this phase. In phase 3 the model is analyzed using analysis tools to generate a controller structure. The parameters of this controller structure is calculated during phase 4, called the design phase. The performance of the system with the generated controller is then compared with the design goals. If the goals are met the process stops, if the goals are not met an iteration is performed, making the designer go through one or more of the previous phases again, changing some parameters.

The first phase of the design process is normally performed by the designer without computer based tools. The control system designer is often faced with a control problem generated by decisions taken by others. Therefore this phase in often not explicitly performed when the process is started, but if the goals can not be met the designer may have to change the design goals. The modelling phase of the design process is seldom considered when describing new analysis or design algorithms. A model in an appropriate domain is used as the starting point. For real systems the modelling phase can be quite difficult, especially when the performance of the system is to be optimal.

Many computer based tools are available to assist the control system designer in phase 3, the analysis phase, and some tools have been made for phase 4, the design phase (e.g MATLAB). However, these tool are linear in the sense that they transform an input such as a model description in some domain to some performance related measure.

The designer is then faced with the problem of evaluating the measure with respect to the specified goals and performing the iteration if necessary. The task of evaluating a design is normally quite complicated and if the design goals are not met it is normally not obvious which actions are best to be taken. This task is not easily done using current CACSD systems. The designer has to perform the iteration himself and determine whether or not the selected design criterion has improved.

4 CACSD Objectives

The objectives described briefly here are Integration, Extendability and Design Actions Support.

Integration is seen as the possibility to integrate different basic software packages such as MATLAB, ACSL, etc. in a way that enables the designer to use the tool best suited for the task without having to go from one model representation to another. The use of currently established software standards in CACE enables the designer to take advantage of new features available in these packages. Further, it should be possible to integrate new packages into the system in a similar way.

By integrating these different basic tools the mental leaps that are normally required when going from one tool to another are reduced thus enabling the designer to have a wider variety of tools available.

Extendability is seen as the being able to extend the current selection of tools with new ones in a simple manner and use these tools in a similar way as the build-in ones. The new tools should be build from already existing tools in the system. Using the model of the design process as a basis some objective for designer support in CACSD system can be formulated. Some form of data and tool management system should be provided, the iteration and evaluation phases of the design model should be supported.

User Support is essential especially when the iterative nature of the design process is taken into account. The user should have tools available for evaluation of the design and analysis results and for doing the iteration. Concepts such as dynamical data access, the concept of analysis results seen as views on the object in question are described in greater detail in [15] and [16]. The user should also have easy access to change to one or more alternate models of the system or alternate controllers. Like in a deck of cards the top card will describe the current system but this card can easily be flipped to evaluate an alternative design or the current design on a more complex model of the system.

5 Software engineering view on the implementation process

The fundamental concept of the software engineering is the project life-cycle, c.f. e.g. [22]. We do not want to recall here the more detailed description of this concept limiting ourselves just to saying that there exist at least three kinds of project life-cycles.

- *information system life-cycle* — starting from formulation of user requirements, going through analysis, design, hardware and software development, installation to the maintenance phase,
- *software life-cycle* — starting when software product is conceived and ending when software product is no longer available for use,
- *software development cycle* — beginning when a software product is approved to develop (improve or maintain) and ending when it is brought to the operational status.

The implementation itself is only one phase within the software development cycle, following the analysis phase. It contains the following stages:

- *program specification* — program functional specification, data structures description and user-interface requirements,
- *program coding* — using top-down or bottom-up approaches, prototyping etc.,
- *program testing* — including debugging and other methods of program quality assurance.

In parallel to all these stages program documentation process should proceed.

For finding the feasible solution of the software project certain factors creating the actual context of the implementation have to be identified in the preceding analysis phase. From the point of view of CACSD requirements, such factors include :

- *the purpose of the program* — training, research, industrial application,
- *the type of the user* — casual, professional, undefined,
- *technical requirements* — the needed response time, the preferred interaction type etc.

The factors important from the point of view of project management are :

- *the estimated project size* — number of code lines, number of terminals etc.,
- *available hardware resources* — mainframes, workstations, PC's, data acquisition systems etc.,
- *available software resources* — previously acquired libraries, packages, the earlier developed code etc.,
- *available financial resources.*

The analysis should result in the answer for the question — who and how will contribute to the project during the software implementation phase.

There are at least three groups involved in the typical CACSD software project life-cycle. Their relations are presented below.

- *ALGORITHM DESIGNERS.*
  Background: Applied mathematics, control theory.
  Objectives: Generality of abstract formulation, correctness of the design approach.
- *SOFTWARE DEVELOPERS.*
  Background: Computer science, software engineering.
  Objectives: System performance, efficiency, reliability, testability.
- *END USERS.*
  Background: Control engineering, process control.
  Objectives: Specific design problem solution.

The real problems appear during the implementation because of the difference of backgrounds and objectives of all the mentioned groups. The increasing specialization is the price which has to be paid for the technological progress. The current software technology makes it almost impossible for the control engineer to write a full-grown commercial quality software product. Although it is regrettable, the more efficient the new software technologies are the more knowledge is required to use them in program implementation, however it does not have to mean that the more knowledge is required to use the final product. It is out of question now that commercial software development is a matter of specialized software companies hiring professional programmers rather than small control research groups. On the other hand, these groups are able to make a substantial progress in the algorithmic development and many products which were later successfully commercialized were born in academic centers. There are different possible ways out. One of them is the so called *concurrent engineering* aiming at the integration of the various teams around the common project goal. The other is creation of the software development environments which will be standardized and integrated enough to allow the non-professional or semi-professional to make the effective use of it. At the moment, generally the gap between the users and software developers has to be accepted as a given fact which cannot be neglected c.f. [11].

6 Developing the original CACSD software

There exist situations when developing the original CACSD may occur preferable. They may concern specific training programs or educational minitools. Sometimes the need for this kind of development may be caused by the lack of software fitting to the configuration of the available resources. A special category are the programs which have to protect the proprietary solutions of the user organization, or other type of the intellectual property, from the possible competition. The last but not least are the products which are planned to be commercialized.

The basic requirement for medium to large size projects is the professional programmers team work. It is very important that inside the team are both algorithm specialists, software developers and "representative" users. It is completely clear that

projects of that scale cannot be successful if they are not well managed.

As far the coding is concerned, there exist well established software engineering methodologies supporting both productivity and quality of programming. Just to recall, the basic principles are :

- *modularity* — the internal design of each program component should be organized in such way that it does not depend on the internal design of any other component,
- *conciseness* — the code should be clear and easy to understand at least by other members of the programming team,
- *structured approach* — large programs should be decomposed into manageable size parts with well-defined relationships,
- *testability* — the program components should be easy to be independently tested and debugged.

All the structural languages like C or Pascal and object-based languages like Ada or Modula-2 do support the programming styles emphasizing the above properties. The basic concept used in structural programming is the *procedure* (function) with data structures hierarchy a bit in the background. There are two extremes in structural programming — the use of "pure" functions with no external dependencies and the whole interface in arguments, and the use of functions "parameterized" by the global variables. The first solution may be very inconvenient, but the second is certainly "less structural" and in fact may break the modularity principle, and result in the code which will be completely not *reusable*, i.e. when it comes to upgrading or integration with another environment the large parts of the program will have to be reorganized if not rewritten.

The above contradiction seems to be resolved in the recently rapidly developing object-oriented programming technology. The basic concept here is the *object* possessing its own data (its state) which are protected (hidden) from the unauthorized use, and its own code, similarly to the classical procedure. The object-oriented approach enhances such programming mechanisms as:

- *data abstraction* — expressing the separation of the program design concepts from the implementation details, hidden (encapsulated) in the objects,
- *inheritance* — allowing to define new object types (classes) on the basis of already defined more general object types by specializing it, i.e. specifying the differences.

The "good style" of object-oriented programs should assure a high level of potential code reusability, c.f. [12]. The object-oriented languages provide certain mechanisms increasing program *expressiveness* like automatic *dynamic memory management*, *polymorphism* — allowing context dependent meaning of certain programming constructs and *late binding* — postponing some code processing from the compile-time to the run-time what may significantly increase program flexibility. There is one more property advocating for the object-oriented approach. It turns out that the way the designer thinks on the model of some reality is itself oriented on objects rather than on functions, c.f. [4], [10].

The recent rapid development of the object oriented languages like C++, Objective C, Smalltalk 80, CLOS and others, reflects the importance of the problem of code reusability. At the moment high expectations accompany the introduction of the new kind of tool, called class libraries, both of general and specialized application, c.f. [3], [8]. They offer a kind of "reusable software components" — a self-contained pieces of code to be used in user applications, allowing the programmer to concentrate on higher level abstractions (e.g. matrices, vectors) and formulate the algorithm in their expressive language, with virtually no loss in the run-time efficiency.

It is hard to overemphasize the role of the object oriented programming technology as the programming productivity tool. There have been already reported certain object-oriented programming applications in the field of CACSD, c.f. [1], [14], [9], but the real "software revolution", c.f. [5] is still to come.

7 CACSD software integration

However, as it was said previously, the acquiring of the ready-made tools is mostly preferable, the typically chosen alternative is adopting available tools to the given application. This is caused by the fact that problems solved in the design practice are rarely "standard", there are always certain their properties which require specific algorithmic improvements or specializations etc. In this way the problem of CACSD tools *extensibility* is raised.

Let us begin with single tool extensibility problem on the MATLAB example, to show what kind of limitations are met here. The MATLAB concept of the programming environment conforms to the traditional, horizontally divided software architecture, see Figure 2.

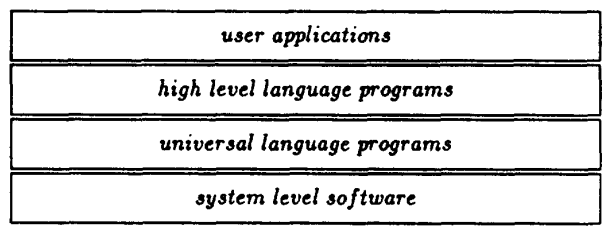

Figure 2: Horizontally divided software architecture.

In the lower level we have internal MATLAB functions, in the higher level Toolbox libraries, and user scripts and functions on top. This architecture is quite logical but suffers from the serious drawbacks. First, the user files are interpreted, or semi-compiled and that degrades their run-time efficiency (there is a "way-out" by constructing executable equivalents, but it is far from being automated and the "ease-of-use" is rather problematic). Second, the user files does not have the "equal access" to program workspace (except for global variables, and interpreted "eval" mechanism). And third, the user cannot construct (and use in MATLAB) other data types than those supported by MATLAB. These limitations are simply consequences of the MATLAB program architecture concept and would be very hard to overcome in "evolutionary way".

The newly introduced Xmath tool which is functionally equivalent to MATLAB, uses the object-oriented programming to cope with the extensibility problems. It offers more rich collection of the elementary data types together with some object creating capabilities (e.g. list) in MathScript — the Xmath's programming language. According to [9], linked user-functions and callable user-interface will be soon available and as contrasted to MAT-

LAB vendors policy, the specification of this interface is planned to be published. If general user defined objects to be used within the Xmath workspace were also available, this environment could become a general platform for CACSD tool integration. Now, it is too early to make such predictions.

A lot of effort was already devoted into creation of multi-tool integrated CACSD environments. However, they may be useful in proprietary settings, there is a common feeling that there is a small hope for making them portable. There also many unanswered questions concerning the concepts of multi-tool integrated environments. Should the integration be model-data or tool-methods oriented ? Should it possess a common "integrated" user-interface or should it rely on separate tools' user-interfaces ? [15], [16] and [1], but it seems to be too early for definitive answers. Although, it is commonly agreed that the evolution of the software systems definitely tends towards the reusable component architecture, see Figure 3, both vertically and horizontally

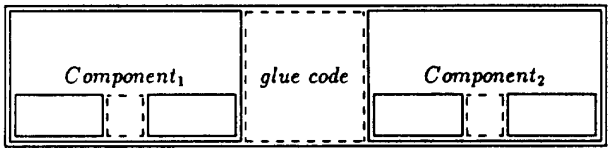

Figure 3: Reusable software component architecture.

divided, with system components ("glue code") supporting the communication (data exchange?) between the components belonging to the given software level. So far, one of the successful steps in this direction is the X-Window system. It is quite probable that the true integration of high level tools is only possible if their lower level components fit well. As long there is no clear accepted standard for the dynamical (run-time) exchange of the structured data (objects) between separate programs, all the integrating attempts are more or less temporary.

Some progress in the creation of first programming systems supplying both object-based programming languages and object abstraction at the operating system level has been already made, c.f. [6]. These kind of systems enable objects to be maintained, managed and used most efficiently, and what seems to be very important from the point of view of CACSD tools integration, they support *object sharing* by independent programs, users etc. Although, the methods of communication (object interaction) are being developed, the new challenge emerges, i.e. the *standardization* of the objects used in the area of CACSD.

8 Conclusion

The software engineering perspective on CACSD tools is described together with some of the background of early CACSD tools. A simple model of the design process has been presented and two areas not easily handled using current CACSD packages are pointed out. They are the evaluation phase and the iteration in the design model. The objectives of integration and extendability and the features of user support for the iteration are described. Taking a software engineering approach some guidelines of developing original CACSD software and the problems of software integration are presented. The problems concerning information system, software life-time and software development cycle are treated and the inherent problem of the different objectives of algorithm designers, software developers and end users is outlined. The importance of current trends in software architecture (object sharing, dynamic data exchange etc.) is pointed out.

References

[1] M.Andersson, S.E.Mattsson, B.Nilsson : On the architecture of CACE environments. *Computer Aided Design in Control Systems - CADCS 91*, Preprints of the 5th IFAC Symposium, Swansea, UK, July 1991, ed. H. A. Barker. Pergamon Press, Oxford 1991, pp. 63–68.

[2] H.A.Barker, M.Chen, P.W.Grant, I.T.Harvey, C.P.Jobling, A.P.Parkman, P.Townsend : The making of eXCes — a software engineering perspective. *Computer Aided Design in Control Systems - CADCS 91*, Preprints of the 5th IFAC Symposium, Swansea, UK, July 1991, ed. H. A. Barker. Pergamon Press, Oxford 1991, pp. 27–32.

[3] G.Booch, M.Vilot : The design of C++ Booch components. *ACM Conference on Object-Oriented Programming, Systems, Languages and Applications / European Conference on Object Oriented Programming OOPSLA/ECOOP '90*, Ottawa, November 1990, pp. 1–11.

[4] B.J.Cox : Object-oriented programming approach — an evolutionary approach. Addison Wesley, Reading 1986.

[5] B.J.Cox : Planning the software industrial revolution. *IEEE Software*, November 1990, pp. 25–33.

[6] R.S.Chin, S.Chanson : Distributed object-based programming systems. *ACM Computing Surveys*, vol 23, no. 1, 1991, pp. 91 - 124.

[7] E.J.Davison (ed.) : *Benchmark problems for control systems design.* Report of the IFAC Theory Committee, Laxenburg, May 1990.

[8] J.M.Dlugosz : Libraries with class. *Byte*, February 1991, pp. 164–168.

[9] M.A.Floyd, P.J.Dawes, U.Milletti : Xmath — a new generation of object-oriented CACSD tools. *First European Control Conference ECC'91*, Grenoble, July 1991, ed. C. Commault et al., vol. 3, Hermes, Paris 1991, pp. 2232–2237.

[10] S.Gossain, B.Andersson : An iterative-design model for reusable object-oriented software. *ACM Conference on Object-Oriented Programming, Systems, Languages and Applications / European Conference on Object Oriented Programming OOPSLA/ECOOP '90*, Ottawa, November 1990, pp. 12–27.

[11] J.Grudin : Interactive systems — bridging the gap between developers and users. *Computer*, April 1991, pp. 59–69.

[12] K.J.Lieberherr, I.M.Holland : Assuring good style for object-oriented programs. *IEEE Software*, September 1989, pp. 38–48.

[13] A.G.J.MacFarlane, G. Grübel, J.Ackerman : Future design environments for control engineering. *Automatica*, vol. 25, no. 2, 1989, pp. 165–176.

[14] J.Parr, P.W.Grant, C.P.Jobling : Object-oriented programming and the implementation of a block diagram editor for the Macintosh. *Computer Aided Design in Control Systems – CADCS 91*, Preprints of the 5th IFAC Symposium, Swansea, UK, July 1991, ed. H. A. Barker. Pergamon Press, Oxford 1991, pp. 194–199.

[15] O.Ravn : On user-friendly interface construction for CACSD packages. *1989 IEEE CSS Workshop on Computer-Aided Control System Design*, December 1989, Tampa, Florida, pp. 35–40.

[16] O.Ravn : Objectives and concepts in computer aided engineering. *Proceedings of the Nordic CACE Symposium*, Technical University of Denmark, Lyngby 1990, pp. 1.11– 1.16.

[17] M.Rimer, D.K.Frederick, C.Y.Huang : Solutions of the second benchmark control problem. *IEEE Control Systems Magazine*, vol.10, no. 4, August 1990, pp. 33–39.

[18] M. Rimvall : *ELCS — the extended list of control software (Swiss edition)*. No. 4, ETH, Zurich, December 1987,

[19] W. Schaufelberger: Educating future control engineers. *Preprints of the 11th IFAC World Congress*, Tallinn, 1990, vol.1, pp. 82–93.

[20] C.Schmid : Techniques and tools of CADCS. *4th IFAC Symposium on Computer Aided Design in Control Systems*, Beijing, PRC, 1988. IFAC Proceedings ser. 1989, no 7, pp. 91–99.

[21] M.Szymkat : Computer aided control systems design. *Computer methods in control engineering*, Ed. W.Wajs, Academy of Mining and Metallurgy, Cracow, 1991. (in Polish)

[22] E.Yourdon : *Managing the system life cycle*. 2nd edition, Yourdon Press, Englewood Cliffs, 1988.